Computer Science • 14 (4) 2013 http://dx.doi.org/10.7494/csci.2013.14.4.667

Tomasz Szydło
Paweł Suder
Jakub Bibro

MESSAGE-ORIENTED COMMUNICATION FOR IPV6-ENABLED PERVASIVE DEVICES

Abstract *An increasing number of electronic devices are being equipped with radio interfaces intended for communication with and control by other devices and applications. Wireless communication in this class of devices may be exposed to a number of situations that can occur, including limited energy resources, equipment failures, node mobility, and loss of communication between nodes. Ultimately, commonly-used standards and protocols for sharing services are not practical and do not take into account the occurrence of such problems. This paper presents a concept of communication that relies on the exchange of messages between wireless-pervasive devices commonly found in our environment.*

Keywords sensor setworks, IPv6, message-oriented middleware

1. Introduction

The majority of pervasive devices installed in our homes, cars, and offices are expected to be embedded with small and low-cost microprocessors integrated with radio transceivers. These devices will be able to offer their functionality as services enabling other devices, and smart applications to interact with them dynamically composing a complex systems. In order to enable interoperability with existing applications and services, several design decisions should be made according to the network communication and service exposition.

There are several middlewares [12] for sensor networks developed to simplify the process of sharing and utilizing functionalities exposed by the devices, but it might be difficult to design applications composed of services provided by heterogeneous devices such as sensor networks, embedded devices and servers due to potential incompatibility of network communication mechanisms. A different approach assumes the usage of the technology used to develop and compose applications in Service Oriented Architecture (SOA) [3]. We think that these devices should use the widely-adopted network communication protocol stack and the service exposition protocols. This will enable the possibility of seamlessly integrating services that are exposed by pervasive devices with other existing services.

Most of the networked devices use the Internet Protocol (IP) suite. This provides interoperability between wired and wireless communication media despite the fact that they express different behaviors in the terms of quality guarantees. Each device that attempts to communicate using the IP protocol stack requires an IP address. It has been predicted that by 2020, there will be 50 billion Internet-enabled devices in the world, so the currently used 4.3 billion IPv4 addresses are insufficient, and the IPv6 protocol is necessary where the address has 128 bits instead of 32 bits. There are a few operating systems for small embedded devices that provide IP connectivity and might be utilized in real world applications.

Service-enabled pervasive devices will rely on error-prone wireless communication technologies that might influence the usage of the services which represent their functionality. Wireless communication between pervasive devices faces several issues, including power consumption constraints for nodes using batteries or energy harvesting, node failures, mobility of nodes, and communication failures. Services implemented using standards such as REST or WS* are typically conveyed using HTTP protocol with XML serialization; thus, clients have to deal with the lack of availability of services, asynchronous invocations, and other problems that might appear more frequently in pervasive devices than in a typical client-server applications. Another approach to such communication relies on the concept of messages. Sending messages across channels decreases the complexity of the end application, thereby allowing the developer of the application to focus on true application functionality instead of the intricate needs of communication protocols. Message-oriented middleware (MOM) [1] is a specific class of middleware supporting the sending and receiving of messages between distributed systems. MOM allows application modules to be distributed over

heterogeneous platforms and reduces the complexity of developing applications that span multiple operating systems and network protocols. The middleware creates a distributed communications layer that insulates the application developer from the details of the various operating systems and network interfaces. Message-oriented middleware may provide reliable asynchronous communication mechanisms that might be used to carry i.e. SOAP messages or other remote communication messages.

The usage of IPv6 by pervasive devices enables the possibility of reusing existing network infrastructure for device communication, while MOM introduces asynchronous communication between entities, providing message caching, reliable communication, and location transparency. Such communication may handle the issues in wireless communication between pervasive devices. Nodes mobility can be handled by the queue and the topic names, while connectivity problems or the sleeping nodes can be handled by the message caching in the message brokers.

In this paper, we propose MOM for IPv6 enabled devices with constrained resources. Section 2 discusses the related work. Section 3 shows the proposed protocol for constrained devices, while section 4 its implementation details for modern operating system designed for constrained resources devices. Section 5 presents the example use case while section 6 sums up the article and discusses future work.

2. Related work

There are several MOM's that differ in details and offered functionality. Two of the most common MOM [4] paradigms are message queuing and publish/subscribe messaging. Message queuing provides buffering between pairs of hosts over point-to-point paths, an approach that is most suitable for applications requiring persistence but less useful for applications requiring responsiveness. In pub/sub messaging, subscribers specify their interest in messages typically by requesting a type of messages or the topic on which they are published. When a publisher sends a message, subscribers who have registered their interest in the message receive it asynchronously. A publish/subscribe model follows a many-to-many communication pattern, allowing for a decoupling between publishers and subscribers, while the queuing model follows a one-to-one model. The architecture of MOM systems includes an additional component which act as the message transfer agents – message brokers. Depending on the technology and implementation message, brokers might be implemented as single hosts or might be federated to improve scalability and high availability of communication.

The Java EE programming environment provides a standard API called Java Message Service(JMS) [8], which is implemented by most MOM vendors and aims to hide the particular MOM API implementations. JMS does not define the format of the messages that are exchanged, so JMS systems are not inter-operable. The Advanced Message Queuing Protocol (AMQP) is an emerging standard that defines the protocol and formats used in the messaging server and client, so implementations are inter-operable. AMQP is defined to provide flexible routing, including common messaging paradigms like point-to-point, publish/subscribe, and request-response. It

also supports transaction management, queuing, distribution, security, management, clustering, federation, and heterogeneous multi-platform support. Message Queue Telemetry Transport (MQTT) is an open-message protocol that enables the transfer of telemetry-style binary data in the form of messages from pervasive devices, along high latency or constrained networks, to a server, or a small message broker. Simple (or Streaming) Text Oriented Message Protocol (STOMP), formerly known as TTMP, is a simple text-based protocol designed to work with MOM. It provides an inter-operable wire format that allows STOMP clients to talk with any Message Broker supporting the protocol. Thus, it is language-agnostic, meaning a broker developed for one language or platform can receive communications from client software developed in another language. There are other message-oriented protocols such as OpenMAMA or DDS, but those presented are the ones that have well-established community and which have proven their reliability in production systems.

3. MOM for IPv6 enabled pervasive devices

Requirements for the message-oriented protocol that might be applied to pervasive devices are different from those for desktop computers. The first requirement is concerned with the simplicity of the messaging protocol, while second concentrates on the transport-layer protocol that must be adequate to the profile of constraint resources devices and wireless connectivity.

The trends in current pervasive development involves the re-usage of web standards to connect the number of embedded devices that appear in the environment. Well-accepted and -understood concepts and standards, such as URI, HTTP or REST, are used to access the functionality of smart objects, mainly due to the self-descriptive semantics of the operations offered by these standards and straightforward implementation. Message-oriented middleware for pervasive devices should fit the current trends, providing easy access to the underlying infrastructure by providing a well-defined programming interface.

Most pervasive electronic devices use wireless communication in the physical layer of the protocol stack. A majority of existing technologies use 2.4GHz ISM band due to its unlicensed access. There are heavy constraints on the data link layer for service-enabled real world devices due to the power efficiency requirements of these devices. Broadly-used wireless standards such as IEEE 802.11 b/g/n do not meet these requirements, so the IEEE 802.15.4 [7] standard has been designed to be supported by semiconductor vendors. Any introduction of IP protocol for such devices must be preceded by several conceptual decisions. In the network layer, IPv4 does not have sufficient addressing space; thus, IPv6 should be used for addressing, as its size is large enough for future usage. The header of the IPv6 frame would fill the major part of energy-efficient data link protocol as IEEE 802.15.4, so there is a need to use the adaptation layer defined in 6LoWPAN [9] specification. Usage of the TCP protocol in the transport layer might introduce unnecessary overhead for communications. On the other hand, usage of UDP protocol might result in the lack of guarantees necessary for

the proper workage of service-orientation protocols, such as WS* and others, based on reliable communication.

Several messaging protocols and message brokers were analyzed, and we finally decided to use the STOMP protocol that is a frame-based protocol, with frames modeled on HTTP. A frame consists of a command, a set of optional headers and an optional body. STOMP is text based, but it also allows for the transmission of binary messages. The default encoding for STOMP is UTF-8, but it supports the specification of alternative encodings for message bodies. It distinguishes itself by covering a small subset of commonly-used messaging operations rather than providing a comprehensive messaging API.

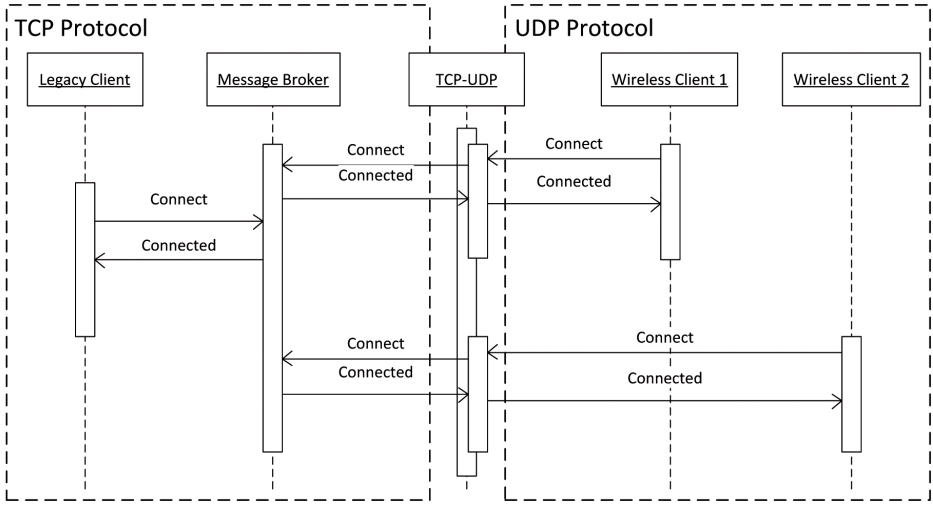

Figure 1. Concept of environment emulation.

The STOMP protocol uses TCP in the transport layer, which is not sufficient for pervasive devices due to the introduced overhead; thus, we decided to introduce a mediator between message broker and the clients, as presented in Figure 1. The *TCP-UDP mediator* is an application that is used to transfer data from UDP datagrams to the appropriate TCP connection. Mapping between the reliable TCP protocol and the unreliable UDP protocol is possible only because STOMP protocol introduces acknowledgments and retransmissions of messages in the application layer. Mapping of the UDP datagram to the TCP connection is based on the source address of the datagram. If there is no mapping, the TCP connection to the broker is established as soon as first UDP datagram is received. This strategy is justified by the fact that devices use IPv6; thus, there is no unambiguous mapping. The next section discusses implementation details of the communication library for embedded devices with constraint resources.

4. Implementation details

There are a few operating systems for small embedded devices that provide IP connectivity. TinyOS [5] is a free and open-source component-based operating system written in the nesC programming language as a set of cooperating tasks and processes. It contains the BLIP library, which is an implementation of a number of IP-based protocols. The second (and most promising) operating system is Contiki, which is an open-source implementation for networked, memory-constrained systems with a particular focus on low-power wireless devices. Examples of where Contiki is used include street lighting systems, sound monitoring for smart cities, radiation monitoring systems, and alarm systems. Contiki provides three network mechanisms: the uIP TCP/IP stack [2], which provides IPv4 and IPv6 networking, and the Rime stack, which is a set of custom lightweight networking protocols designed specifically for low-power wireless networks. The IPv6 stack was contributed by Cisco and was, at the time of release, the smallest IPv6 stack to receive IPv6-Ready certification.

Contiki OS is not multi-threading, but it provides the `protothreads` library which uses an event-driven mechanism built into the system. Events are sent by the processes to the dispatcher which notifies other processes of occurred events. It is possible that the process waits for an event from another process, enabling inter-process communication. The template for blocking functions used in our implementation is presented in Listing 1. This allows synchronization of the processes performing related tasks.

Listing 1: Template for blocking functions for Contiki OS

```
#define STOMP_<COMMAND>(<arg>, ...)
    stomp_<command>(<arg>, ...);
    PROCESS_WAIT_EVENT_UNTIL(ev == PROCESS_EVENT_CONTINUE);
```

The network communication library for Contiki extensively uses eventing mechanisms. Sending and receiving data should be done in the same process which initiated the connection. The networking library handles several TCP states, while in UDP communication there is only information about number of bytes to send. It is important to strictly follow the introduced pattern because the communication is based on the shared memory buffers, and the data is not copied between memory blocks.

The implemented `stomp` library for Contiki OS provides the API that follows the STOMP 1.1 specification and handles the UDP communication as discussed in the previous section. It allows applications to connect to the message broker, send a message to a specific queue, manage subscriptions queues to receive messages, and confirm and report operations in the transaction. Listing 2 shows the subset of implemented API, while Listing 3 the example usage of the library. The prototype implementation meets significant resource constraints as it is characterized by a small memory footprint. The size of the core Contiki OS is about 50kB of flash memory, while the `stomp`

library footprint is about 15kB. This means that, for the typical micro-controller with 128kB of flash memory, 60kB is left for user application.

Listing 2: Subset of `stomp` library API

```
#define STOMP_CONNECT(host, login, pass)
#define STOMP_SUBSCRIBE(id, destination, ack)
#define STOMP_SEND(destination, type, length,
                           receipt, tx, message)
#define STOMP_ACK(subscription, message_id, tx)
#define STOMP_NACK(subscription, message_id, tx)
```

Listing 3: Example usage of the `stomp` library

```
PROCESS_THREAD(stomp_client_process, ev, data) {
    PROCESS_BEGIN();
    register_callback();
    STOMP_NETWORK_CONNECT(&ipaddr, port);
    forever {
        if (not connected) {
            STOMP_CONNECT("apollo", "admin", "password");
            STOMP_SUBSCRIBE("income", "/queue/"+UUID, "auto");
            for (sensor in sensors) {
                STOMP_SUBSCRIBE("income"+sensor.name,
                                "/queue/"+sensor.name, "auto");
            }
            STOMP_SEND("/queue/manager", "text/plain",
                       NULL, NULL, NULL,
                       "HELLO "+UUID+" temp,hum,pres);
        } if (connected) {
            wait(15s);
            STOMP_BEGIN("tx");
            //internal operations
            STOMP_SEND(...);
            STOMP_COMMIT("tx");
        }
    }
    PROCESS_END();
}
```

In the prototype, the *TCP-UDP mediator* has been implemented as a multi-threaded Java program. During the development, we used an Apache Apollo message broker that is based on ActiveMQ. Apollo is a multi-protocol broker and supports

STOMP, Openwire, MQTT, SSL, and WebSockets. In our solution, the *TCP-UDP mediator* has been deployed on the same server as Apollo.

5. Use Case

We have designed and developed a sample application for testing purposes. It consists of *Manager Application* and *Sensor Node Application* that might be installed on wireless devices equipped with various sensors. The manager is able to discover wireless nodes and sensor types, configure wireless nodes, and collect measured data from sensors (e.g., temperature and humidity). There are several different design approaches for such an application [11, 6, 10], but none of them uses message-oriented communication. In our case, the wireless communication issues are handled by the messaging middleware that is easily accessible by software clients implemented using various programming languages. We have implemented manager application using `Java` programming language, and the sensor node application in `C` for Contiki OS.

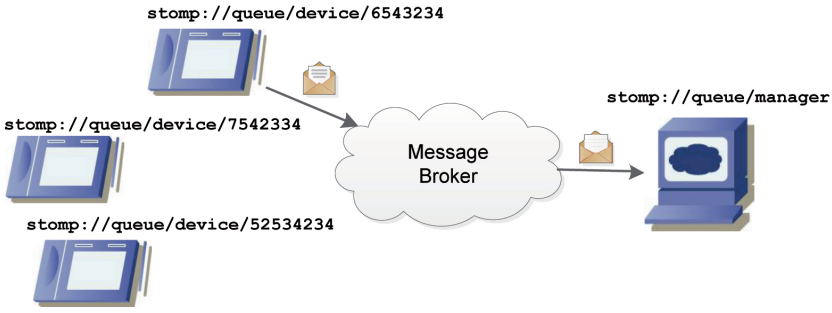

Figure 2. Usage of the MOM in the example application.

The application uses a simple text-based protocol developed to describe the operations and measured values that are transmitted using MOM. There are four kinds of messages used for exchanging information between nodes and manager application: `HELLO` (advertise presence), `GET` (last measurement), `UPDATE` (take a new measurement), and `SET` (configure device). The application sends (publishes) a measurement request via message broker on a specific device queue on which the device is subscribed. Then, sensor nodes receive the message, performs a task, and replies with a measured value, placing the message on the manager queue as depicted in Figure 2. Then, the application receives the message and extracts the included information. The main window of *Manager Application*, as presented in Figure 3, splits into two panels:

- *Nodes list* – after running the application, a tree-panel shows up on the left. It consists of available wireless nodes and provides additional information such as device queue name – queue is responsible for gathering messages published by *Manager Application*, sensors list – list of available sensors (e.g. temperature,

pressure, pollution meter etc.), last successful measurement, update parameter – determines situations when manager should to receive a message (i.e. always, on change), and time intervals. The list is updated any time a new wireless node appears.

- Visualization panel – is split into tabs according to available sensor nodes. For every node, it displays graphs showing measurements obtained from different sensors. It is refreshed when new measurement message are received and a new tab is added if a new wireless node appears.

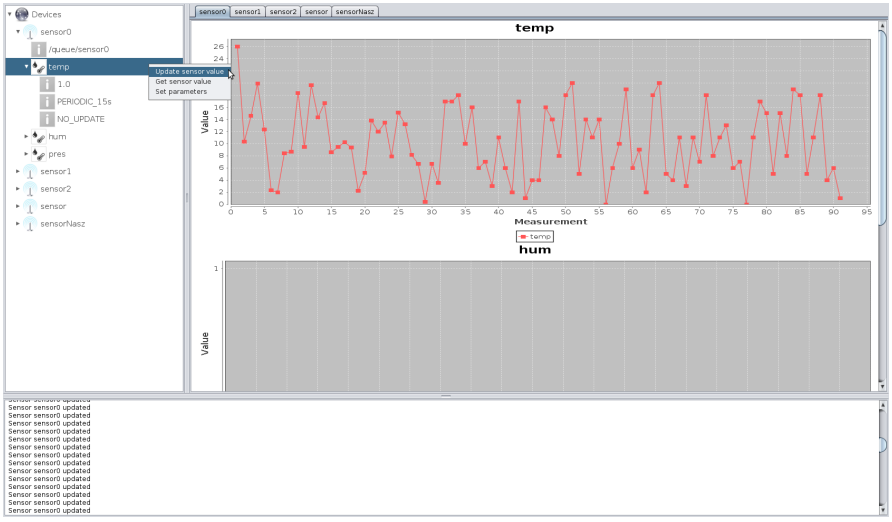

Figure 3. Screenshot of Manager Application.

For concept-verification purposes, we have developed a simple wireless device based on a Zigduino circuit board, which incorporates a very-low-cost ATmega128RFA1 microcontroller and IEEE 802.15.4 radio. As a router for IPv6 and 6LowPAN communication, we have used a Linux-based device equipped with an additional network card driven by STM32W108 micro-controller and customized Contiki OS image.

Figure 4 shows the results of the benchmarking test which we performed. Messages have been sent from the single wireless device to the manager. The throughput is lower than the bandwidth of the IEEE 802.15.14 due to the fact that construction of 6LowPAN frame and STOMP is time-consuming for the 8-bit microprocessor we used. Nevertheless, the acquired efficiency is satisfactory for most embedded applications.

6. Conclusions

In this paper, we have presented the concept of message-oriented communication for IPv6 enabled pervasive devices. It provides reliable communication mechanisms that might be used for service provisioning that express functionality of pervasive devices.

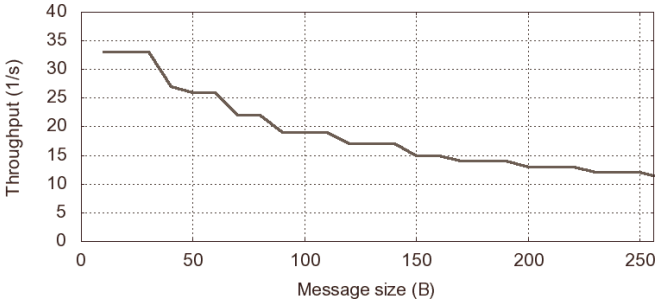

Figure 4. Results of benchmarking tests.

The concept has been verified by the sample application deployed on the physical devices. The small footprint of the library enables its possibility to be used in various applications. The work might be further extended of discoverability mechanisms, simplifying the initial configuration of the system. A message broker could be discovered on the network using one of the service discoverability protocols such as *uPnP* or *SLP* in order to provide address of the message broker.

Acknowledgements

The work reported in this paper was supported by the AGH-UST grant no. 15.11.120.263.

References

[1] Curry E.: Message-Oriented Middleware. In: Q. H. Mahmoud, ed., *Middleware for Communications*, chap. 1, pp. 1–28. John Wiley and Sons, Chichester, England, 2004. ISBN 978-0-470-86206-3.

[2] Dunkels A.: The uIP Embedded TCP/IP Stack The uIP 1.0 Reference Manual, 2006.

[3] Erl T.: *Service-Oriented Architecture: Concepts, Technology, and Design*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2005. ISBN 0131858580.

[4] Kim M., Karenos K., Ye F., Reason J., Lei H., Shagin K.: Efficacy of techniques for responsiveness in a wide-area publish/subscribe system. In: *Proceedings of the 11th International Middleware Conference Industrial track*, Middleware Industrial Track '10, pp. 40–45. ACM, New York, NY, USA, 2010. ISBN 978-1-4503-0456-6. `http://dx.doi.org/10.1145/1891719.1891726`.

[5] Levis P., Madden S., Polastre J., Szewczyk R., Whitehouse K., Woo A., Gay D., Hill J., Welsh M., Brewer E., Culler D.: TinyOS: An operating system for sensor networks. In: *Ambient Intelligence*. Springer Verlag, 2004.

[6] Madden S. R., Franklin M. J., Hellerstein J. M., Hong W.: TinyDB: an acquisitional query processing system for sensor networks. *ACM Trans. Database Syst.*,

vol. 30(1), pp. 122–173, 2005. ISSN 0362-5915. http://dx.doi.org/10.1145/1061318.1061322.

[7] Man L. A. N., Committee S.: IEEE Standard for Information technology- Telecommunications and information exchange between systems- Local and metropolitan area networks – Specific requirements – Part 15.4: Wireless MAC and PHY Specifications for Low-Rate WPANs. *Control*, vol. 2006(September), p. 323.

[8] Microsystems I. S.: *Java Message Service, Version 1.0.2 (JMS specification)*. Tech. rep., Sun Microsystems, Inc., 1998. http://java.sun.com/products/jms.

[9] Montenegro G., Kushalnagar N., Hui J., Culler D.: Transmission of IPv6 Packets over IEEE 802.15.4 Networks. RFC 4944 (Proposed Standard), 2007. http://www.ietf.org/rfc/rfc4944.txt.

[10] Muller R., Alonso G., Kossmann D.: *SwissQM: Next Generation Data Processing in Sensor Networks*. In: *CIDR 2007, Third Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 7–10, 2007, Online Proceedings*, pp. 1–9. 2007. http://dx.doi.org/http://www.cidrdb.org/cidr2007/papers/cidr07p01.pdf.

[11] Niec M., Pikula P., Mamla A., Turek W.: Erlang-based sensor network management for heterogeneous devices. *Computer Science*, vol. 13(3), pp. 139–151, 2012.

[12] Raychoudhury V., Cao J., Kumar M., Zhang D.: Middleware for pervasive computing: A survey. *Pervasive and Mobile Computing*, 2012. ISSN 15741192. http://dx.doi.org/10.1016/j.pmcj.2012.08.006.

Affiliations

Tomasz Szydło
AGH University of Science and Technology, Krakow, Poland, tszydlo@agh.edu.pl

Paweł Suder
AGH University of Science and Technology, Krakow, Poland, suder@agh.edu.pl

Jakub Bibro
AGH University of Science and Technology, Krakow, Poland, bibro@agh.edu.pl

Received: 2.03.2013
Revised: 30.05.2013
Accepted: 2.06.2013